공학석사학위논문

정적 분석에 기반한 난독화 기술 비교

Obfuscator Evaluation System
based on Static Analysis

2015 년 8 월

서울대학교 대학원

전기.컴퓨터 공학부

박 지 순

공학석사학위논문

정적 분석에 기반한 난독화 기술 비교

Obfuscator Evaluation System
based on Static Analysis

2015 년 8 월

서울대학교 대학원

전기.컴퓨터 공학부

박 지 순

정적 분석에 기반한 난독화 기술 비교

Obfuscator Evaluation System
based on Static Analysis

지도교수 이 광 근

이 논문을 공학석사 학위논문으로 제출함

2015 년 6 월

서울대학교 대학원

전기.컴퓨터 공학부

박 지 순

박지순의 공학석사 학위논문을 인준함

2015 년 7 월

위 원 장	김명수	(인)
부위원장	이광근	(인)
위 원	신영길	(인)

요약

정적 분석에 기반하여 난독화기를 비교평가하는 방법을 제안한다. 상업용 소프트웨어를 만드는 개발자들은 그들이 개발한 프로그램이 역공학(Reverse Engineering) 기법으로 분석되는 것을 막기 위해 프로그램 난독화기를 이용한다. 좋은 난독화기를 선정하여 사용하기 위해서는 난독화기들을 평가할 수 있는 방법이 필요하다. 이 논문에서는 난독화된 프로그램에 대해 의미기반 정적 분석을 시도하고 분석의 결과를 비교하는 방법을 통해 다양한 난독화기를 간단히 평가할 수 있는 방법을 제안한다. 그리고 실험을 통해, 제안된 방법으로 얻어진 평가 결과가 이미 알려진 난독화 기술들 간의 우열과 일치함을 보인다.

주요어: 정적 분석, 프로그램 분석, 난독화, 복잡도 평가

학번: 2013-23116

목차

그림 목차

표 목차

제 1 장 서론

1.1 동기

효과적인 프로그램 난독화를 위해서는 적절한 난독화기의 선정이 선행되어야 한다. 상업적인 프로그램을 개발하는 개발자들은 역공학 기법을 통해 프로그램의 로직 또는 데이터가 유출되는 것을 방지하기 위해 프로그램 난독화기를 사용한다. 일반적으로, 여러 난독화 기술을 조합할수록 더 강력한 난독화가 가능하지만 그만큼 난독화된 프로그램의 성능이 하락한다. 그러므로, 프로그램을 난독화 하려는 개발자는 시중의 다양한 난독화기들 중에 적절한 난독화기를 선정할 수 있어야 한다.

난독화기의 선정을 위해서는 난독화 기술의 영향을 총체적으로 비교평가할 수 있는 방법이 필요하다. 최소한의 성능 하락으로 최대한 강력한 난독화를 제공하기 위해 난독화기 제조사들은 다양한 난독화 기술들을 적절히 조합하는 방법을 사용한다. 그러므로 난독화기를 평가하기 위해서는 각각의 난독화 기술과 이들의 조합을 총체적으로 평가할 수 있는 평가 방법이 필요하다. 현존하는 방법들은 대부분 프로그램의 구문(syntax)에 대한 분석을 통해 난독화 기술을 평가한다. 이런 방법들은 프로그램의 길이, 조건문의 수, 클래스 상속 관계의 복잡도, 제어 흐름의 복잡도 등 단편적인 기준으로 난독화 기술을 평가하기 때문에 다양한 난독화 기술들과 그들의 조합을 총체적으로 평가하는데에는 한계가 있다.

1.2 기존 평가방법들의 한계

난독화 기술을 평가하기 위한 기존의 방법들을 소개하고 그 한계에 대해 설명한다. 다양한 난독화 기술들과 이를 활용한 난독화기들이 발표됨에 따라 난

독화의 영향을 측정하기 위한 방법들도 다수 제안되었다. 이 방법들은 대부분 소프트웨어 공학에서 사용되는 표준적인 측정 방법에 기반하고 있으며, 대표적인 측정 기준으로는 프로그램의 크기, 순환 복잡도(cyclomatic complexity), 자료 구조의 복잡도 등을 들 수 있다[4]. 이런 방법들은 프로그램에 대한 단편적인 구문 분석이나 제한된 의미 분석에 기반하기 때문에 다양한 난독화 기술들을 평가하는데에는 한계가 있다. 특히, 여러가지 난독화 기술들이 동시에 적용된 경우에 다양한 난독화의 효과를 제대로 평가하지 못하므로 시중의 난독화기를 비교하는데 사용하기에는 적절하지 않다. 그림 1.1은 데이터 난독화 기법 중 가장 널리 쓰이는 변수 변환(variable transformations)[1] 난독화 방법을 나타내는 의사 코드이다. 순환 복잡도 또는 엔트로피(entropy)를 기반으로 프로그램의 복잡도를 평가하는 방법[4, 5]은 난독화의 영향을 측정하는 대표적인 방법들이지만, 데이터 난독화를 평가하기에는 적합하지 않다. 예시의 난독화된 프로그램에서, 제어 흐름 그래프 상의 노드와 변의 수는 증가하였지만 경로의 수는 그대로이기 때문에 난독화 적용 후에도 순환 복잡도와 엔트로피는 그대로이다. 데이터의 흐름을 분석하여 데이터 난독화의 복잡도를 측정하는 방법도 있지만, 이 방법은 반대로 제어흐름 난독화 방법을 평가하지 못한다.

1.3 해결을 위한 아이디어

다양한 난독화 기술의 궁극적인 목적은 프로그램에 대한 분석을 방해하는 것이다. 난독화 기술들은 감추고자 하는 대상과 기본 아이디어가 각각 다르다. 예를 들면, 어떤 난독화 기술들은 프로그램 내의 데이터를 숨기기 위한 방법을 제공하고, 또 다른 난독화 기술들은 프로그램의 제어 흐름을 숨기는 방법을 제공한다. 숨기고자 하는 대상이 동일한 난독화 기술들도 그 대상을 숨기는 방법에 따라서 다시 세분화 될 수 있다. 하지만 보다 추상화된 관점에서 난독화 기술들을 바라보았을 때, 이 기술들의 궁극적인 목적은 난독화된 프로그램에 대한 분석을

방해하는 것이다.

그러므로, 난독화를 적용했을 때 프로그램에 대한 분석이 어려워질수록 더 좋은 난독화 기술이라고 볼 수 있다. 여기서 말하는 분석은 단순히 프로그램의 구문으로부터 통계적인 복잡도를 산출해 내는 것이 아니라, 의미 기반 분석(semantic-based analysis)을 통해 프로그램이 갖는 의미를 파악하는 안전한 정적 분석을 뜻한다. 난독화로 인해 프로그램에 대한 정적 분석의 결과가 부정확해진다면 그 난독화 기술은 프로그램에 대한 분석을 어렵게 하는 좋은 기술이라고 생각할 수 있다. 그렇기 때문에, 분석 결과의 정확도의 변화를 정량화 함으로써 난독화 기술 상호간의 우열을 비교하는 것이 가능하다.

이 연구에서는 정적 분석에 기반한 난독화 기술 평가 방법을 제안한다. 이 방법은 정적 분석 기술을 동원하여 원본 프로그램과 난독화된 프로그램을 각각 분석하고, 분석 결과의 변화량을 기반으로 난독화 기술이 프로그램에 미치는 영향을 평가한다. 이 방법을 이용하면 기존의 방법들로는 불가능했던 다양한 난독화 기술에 대한 총체적인 평가가 가능해진다.

1.4 논문 구성

이 논문은 다음과 같이 구성된다. 2장에서는 정적 분석을 바탕으로 난독화 기술을 평가하기 위한 시스템을 설명한다. 3장에서는 평가 시스템을 구성하는 각 요소들에 대해 설명하고 실험 결과를 보인다. 4장에서는 이 연구의 의의와 한계에 대해 고찰하고 5장에서 결론을 내린다.

그림 1.1 전형적인 데이터 난독화의 예

```
1  /* Original Code */
2
3  add3(arg)
4    incr = 3
5    res = incr + arg
6  RETURN with res
```

```
1  /* Obfuscated Code */
2
3  //transform function f
4  f(a)
5    res = pow(a, 7) mod 33
6  RETURN with res
7
8  //inverse transform function g
9  g(b)
10   res = pow(b, 3) mod 33
11 RETURN with res
12
13 add3(arg)
14   incr = 9        //f(3) = pow(3, 7) mod 33 = 2178 mod 33 = 9
15   res = f(g(incr) + g(arg))
16 RETURN with res
```

제 2 장 난독화 기술 평가 시스템

2.1 평가 시스템 설계

본 연구에서 제안하는 난독화 기술 평가 시스템은 프로그램 분석기와 샘플 프로그램, 난독화기의 세 가지 요소로 구성된다. 먼저 난독화가 적용된 샘플 프로그램을 분석하고, 분석결과에 기반하여 난독화 기술을 평가하는 방식이다. 평가자는 평가의 목적 및 방향에 따라 분석기 또는 분석기의 설계를 변경하거나 샘플 프로그램을 대체하는 것이 가능하다.

난독화 기술의 비교평가를 위해서는 무엇보다도 먼저 좋은 난독화 기술에 대한 정의가 필요하다. Xuesong Zhang 등은 [2]에서 난독화 기술들을 프로그램 심벌 정보 난독화(lexical obfuscation), 데이터 난독화(data obfuscation), 제어 흐름 난독화(control obfuscation) 등으로 분류하였다. 이런 다양한 종류의 난독화 기술들을 하나의 지표로 평가하기 위해서는 평가의 기준이 명확하고 총체적이어야 한다. 그러므로, "총체적으로 좋은 난독화 기술"이 먼저 정의되어야 한다. 이 정의 없이는 평가 결과가 어떤 의미를 갖는지 설명할 수 없다.

이 연구에서는 프로그램의 실행 의미(operational semantics)를 더 복잡하게 하는 난독화를 더 좋은 난독화로 정의한다. 소프트웨어 난독화는 프로그램의 궁극적인 의미(denotational semantics)를 유지하면서 실행 의미를 복잡하게 하여 프로그램을 역공학 분석 기술로부터 보호하는 것이다. 그러므로, 프로그램의 실행 의미를 더 복잡하게 하는 난독화 기술이 그렇지 못한 난독화 기술보다 더 좋은 난독화 기술이라고 볼 수 있다. 이 논문에서는 프로그램의 실행 의미에 영향을 주지 못하는 난독화 기술들을 의미없는 난독화로 간주한다. 이후의 내용에서 프로그램의 의미는 실행 의미를 뜻한다.

프로그램 분석 기술을 사용하면 프로그램의 의미의 복잡도를 평가할 수 있다. 프로그램이 복잡해 질수록 프로그램에 대한 분석은 더 어려워지고 분석의 정확도는 떨어진다. 그렇기 때문에 난독화된 프로그램들에 대한 분석결과를 살펴봄으로써 난독화 기술이 프로그램에 미치는 영향을 정량적으로 평가하는 것이 가능하다. 주어진 프로그램에 대해 역공학 분석을 시도할 때 분석이 용이한지 판단하는 기준은 크게 두 가지이다. 첫번째는 분석에 걸리는 시간이고, 두 번째는 분석으로 알아낸 정보의 명확성이다. 이 두 가지 요소를 고려하면 분석 결과를 기반으로 프로그램의 복잡도를 평가할 수 있다.

난독화 기술에 대한 의미있는 평가를 위해서는 안전한 정적 분석 기술이 요구된다. 난독화 기술이 미치는 영향을 평가하기 위해서 원본 프로그램과 난독화된 프로그램 각각에 대한 분석 결과를 관찰하고 분석 결과의 변화량을 계산한다. 이때 프로그램 분석에 사용된 분석기가 안전하지 않은 분석 결과를 제공한다면 두 분석 결과간의 차이가 프로그램이 복잡해진 정도를 잘 반영한다고 할 수 없다. 난독화를 적용할 때 원본 프로그램의 궁극적인 의미는 보존되어야 하기 때문에 난독화된 프로그램에 대한 분석 결과도 원본 프로그램에 대한 분석 결과를 포섭해야 한다. 동적 분석 기술만으로는 안전한 분석을 할 수 없기 때문에 평가 결과가 의미를 가지려면 안전한 정적 분석기를 이용해서 평가를 진행해야 한다.

더 정확한 분석은 더 정확한 평가를 가능하게 한다. 만약 분석기가 충분히 정확하지 않다면 원본 프로그램과 난독화된 프로그램에 대한 분석 결과의 차이를 분별하기 어렵게 된다. 예를 들어, 모든 메모리 요소들을 “알 수 없음”으로 분석하는 분석기는 안전한 분석 결과를 제공하지만 난독화의 평가에는 전혀 쓸모가 없다.

실험을 통해 정적 분석에 기반한 평가 시스템이 더 좋은 난독화 기술을 판별해낼 수 있음을 보인다. 다음 장에서는, 이 평가 시스템을 이용한 평가에서 더 강력한 난독화 기술이 약한 난독화 기술보다 더 높은 점수를 받고, 프로그램의

의미에 영향을 주지 못하는 난독화 기술들이 낮은 점수를 받게 됨을 보일 것이다. 이 평가 결과는 난독화 기술이 정적 분석을 통해 평가될 수 있음을 의미한다.

2.2 복잡도 점수의 계산 방법

난독화 기술의 복잡도 점수는 다음과 같이 계산된다. 난독화 기술의 복잡도를 정량화 하기 위해 먼저 난독화 기술들과 그 조합들을 각 샘플 프로그램에 적용하고 정적 분석을 수행하여 분석 결과를 관찰한다. 준비된 n개의 샘플 프로그램들 $\{P_1, P_2, \cdots, P_n\}$과 l 개의 난독화 기술들 $\{O_0, O_1, \cdots, O_l\}$, 그리고 o개의 분석 항목 $\{A_1, A_2, \cdots, A_o\}$이 있을 때, 관찰된 분석 결과 항목들을 m_{ijk}로 나타낼 수 있다. 이는 샘플 프로그램 P_i에 O_j 난독화 기술을 적용하고 정적 분석하여 A_k 항목을 측정하였음을 의미한다. 여기서 O_0는 원본 프로그램 그대로를 돌려주는 빈 난독화이다.

난독화 기술의 복잡도 점수를 계산하기 위해서는 먼저 각 측정 항목에 대해 원본 프로그램에 대한 분석 결과 대비 증가율을 계산해야 한다. 난독화의 적용으로 인해 분석의 성능이 얼마나 안좋아지는지를 확인하기 위해 각 항목에 대한 증가율의 계산이 필요하다. 증가율 I_{ijk}는 다음과 같이 계산된다.

$$I_{ijk} = \frac{m_{ijk}}{m_{i0k}} - 1$$

증가율을 기반으로 표준점수(Z-score) Z_{ijk}와 편차치(T-score) T_{ijk}를 계산할 수 있다.

$$Z_{ijk} = \frac{I_{ijk} - AVG_{ijk}}{\sigma_{ijk}},$$

$$T_{ijk} = (Z_{ijk} \times 10) + 50$$

여기서 AVG_{ijk}와 σ_{ijk}는 각각 I_{i1k} 부터 I_{ilk} 까지의 항목들에 대해 산술 평균과 분산을 구하는 함수이다. 난독화 기술이 각 분석 방법에 주는 영향을 공평하게

정규화(normalization) 하기 위해 표준 점수가 사용되었고, 표준 점수간의 편차를 한눈에 알아보기 쉽도록 편차치로 변환하였다.

하나의 샘플 프로그램 P_i에 대해서 각 난독화 기술 O_j가 획득한 복잡도 점수는 편차치의 합으로 구할 수 있다. 편차치는 각 분석 항목에 대해 구해진 값이고, 복잡도는 이 항목들을 모두 고려해야 하므로 편차치들을 합산하였다. 편차치들은 이미 정규화 된 값들이기 때문에 단순한 합산으로 총점을 계산하는 것이 가능하다. 평가자가 항목별로 중요도가 다르다고 판단한 경우에는 항목별 가중치를 곱하여 합산할 수 있다.

$$Score_{ij} = \sum_{k=1}^{4} T_{ijk}$$

최종적으로, 각 난독화 기술의 효과에 대한 평가 점수는 샘플 프로그램별 평가 점수의 평균으로 나타낸다. 각 샘플 프로그램이 갖는 특성에 따라 생길 수 있는 평가의 오차를 최소화 하기 위해 다양한 프로그램에 대한 측정 결과에 대한 평균을 이용한다.

$$Score_j = \frac{1}{l} \sum_{i=0}^{n} Score_{ij}$$

제 3 장 구현 및 실험

이 장에서는 실험의 설계와 결과에 대해 설명한다. 실험을 위해 미리 선택된 샘플 프로그램들에 대표적인 난독화 기술들을 적용하고, 정적 분석기를 이용하여 원본 프로그램과 난독화된 프로그램을 각각 분석하여 분석의 결과를 관찰하였다. 그리고 분석 결과를 바탕으로 각 난독화 기술의 영향을 정량화하여 비교해 보았다. 각 절에서는 난독화 기술 평가 시스템의 세 가지 요소들이 어떻게 선택되었는지와 실험의 결과에 대해 각각 설명한다.

3.1 평가 시스템 구성 요소들의 선정

3.1.1 정적 분석기

실험에서는 정적 분석기 Sparrow를 이용해서 프로그램을 분석한다. Sparrow는 요약 해석(abstract interpretation)[9] 기술을 바탕으로 C 소스 코드에 대해 안전한 정적 분석을 수행하는 정적 분석기이다. 이 분석기는 다양한 분석 옵션들을 제공하는데, 이 실험에서는 Dense, Sparse, Selective 분석 옵션을 사용한다. Dense 분석은 요약 해석에 기반한 가장 기본적인 분석 옵션으로, 실험에 사용된 Dense 분석은 지역화(localization)를 이용한 최적화가 적용되어 있다[6]. Sparse 분석은 프로그램을 분석할 때 각 지점에서 분석에 고려되어야 하는 메모리를 식별하고 최소한의 메모리만 분석에 동원함으로써, 분석의 정확도를 떨어뜨리지 않고도 매우 적은 비용으로 프로그램을 분석할 수 있는 진보된 분석 기술이다[7]. Selective 분석은 고비용의 분석 방법을 적용했을 때 더 정확한 분석 결과를 얻을 수 있는 부분을 예비 분석을 통해 식별해냄으로써 약간의 비용만으로도 훨씬 더 정확한 분석 결과를 얻을 수 있는 분석 방법이다[8]. 난독화가 기본적인 분석에

미치는 영향 뿐만 아니라 개선된 우수한 분석 기술들에 갖는 내성도 동시에 평가하기 위해 위의 분석 방법들을 채용하였다.

실험에서는 다음의 항목들을 측정하였다.

- Dense 분석을 했을 때의 분석 소요 시간(초)(A_1)
- Dense 분석 결과의 래티스 원소 높이(Lattice Element Height)(A_2)
- Sparse 분석을 했을 때의 분석 소요 시간(초)(A_3)
- Selective 분석 결과의 래티스 원소 높이(A_4)

3.1.2 샘플 프로그램

난독화 기술의 평가를 위한 샘플 프로그램들은 프로그램의 크기와 특성을 바탕으로 선택된다. 프로그램의 크기가 너무 작거나 특별한 특성을 갖는 프로그램에는 난독화의 효과가 제한적으로 적용될 수 있다. 예를 들면, main 함수 하나만으로 구현된 프로그램은 함수 호출 과정을 복잡하게 하는 난독화 기술을 적용해도 복잡도가 증가하지 않는다. 반대로, 프로그램의 크기가 극단적으로 큰 경우에는 정적 분석에 너무 오랜 시간이 걸릴 수 있으므로 원활한 평가를 위해서는 샘플 프로그램의 선정에 프로그램의 크기를 고려해야 한다.

이 실험에서는 오픈소스 리눅스 라이브러리 코드들 중 분석에 너무 오랜 시간이 걸리지 않는 프로그램들을 선정하였다. 오픈소스 리눅스 라이브러리들은 오랜 시간에 걸쳐 수많은 프로그래머들에 의해 다듬어져왔기 때문에 다양한 프로그램 패턴이 적용되어 있다. 샘플 프로그램이 갖는 특성이 평가에 주는 영향을 가능한 한 최소화 하기 위해 복수의 프로그램들을 선정하였다. 선정된 프로그램들은 다음과 같다.

표 3.1 샘플 프로그램 목록

Program	LOC
bbe-0.2.2	7822
dancer-xml-0.8.2.1	21813
xfpt-0.07	9089

3.1.3 난독화기

실험을 위해 C 소스 코드 난독화기를 직접 구현하여 사용하였다. Sparrow가 C 소스코드 분석기이기 때문에 원본 C 소스코드를 난독화된 C 소스코드로 만들어주는 난독화기가 필요하다. 이 실험에서 사용된 난독화기는 다양한 난독화 기술들을 선택적으로 적용할 수 있도록 CIL 라이브러리[12]를 이용하여 Ocaml 언어로 구현되었다. 이 난독화기는 입력으로 받은 C 소스코드를 CIL 라이브러리를 이용하여 CIL 자료 구조로 변환한다. 적용하고자 하는 난독화 기술들을 이용하여 이 자료구조를 변경한 후 다시 C 소스 코드 형태로 출력하여 난독화된 소스 코드를 획득한다.

대표적인 난독화 기술들을 선정하여 이 난독화기에 구현하였다. 이들은 다수의 난독화기에 공통적으로 채용된 난독화 기술들로, 심벌 정보 변경, 데이터 난독화, 함수 내의 제어흐름 난독화, 그리고 함수 호출 과정에 대한 제어흐름 난독화 등 다양한 방식의 난독화 기술들 중 대표 기술을 선정하였다. 선정된 난독화 기술들은 아래와 같다.

Symbol substitution(SS) 프로그램에 등장하는 지역 변수 및 전역 변수의 이름을 변경하는 난독화이다. 프로그램에 등장하는 변수들의 이름은 때때로 중요한 정보를 포함하고 있을 때가 많다. 예를 들면, 암호화를 수행하는 함수에서 key라는 이름의 변수는 민감한 정보를 담고 있을 가능성이 크다. 변수명을 치환하는 난독화는 프로그램의 모든 변수들의 이름을 의미없는 이름으로 바꾼다. 이를 통해

소스 코드의 가독성을 떨어뜨리는 것이 이 난독화의 목적이지만, 이 연구에서는 프로그램의 의미에 영향을 끼치지 못하기 때문에 의미없는 난독화로 간주한다.

Alternation of string expressions(ASE) 이 난독화는 프로그램에 등장하는 문자열의 표현을 ascii 값으로 바꾼다. 소스 코드에 "Hello world\n"라는 문자열이 등장한다면, 이를"\x48\x65\x6c\x6f\x20\x57\x6f\x72\x6c\x64\x0a"로 치환한다. 이는 SS와 마찬가지로 소스 코드의 난독성을 떨어뜨리지만, 프로그램의 의미와는 무관하기 때문에 이 연구에서는 의미없는 난독화로 간주한다.

Integer transform(IT) 대표적인 데이터 난독화 기술중 하나로서, 프로그램 실행중에 정수 값들이 메모리에 그대로 노출되지 않도록 변환과 역변환을 반복한다. 가장 전형적인 정수 변환 난독화 기술을 그림 1.1에서 볼 수 있다. 이 실험에서는 변환 함수와 역변환 함수를 별도로 구현하는 대신 정적 분석 결과가 "알 수 없음"이 되도록 하는 항등 함수를 구현하여 사용하였다.

Control flow flattening(CFF) 명령들의 실행 순서를 알아보기 어렵도록 프로그램을 변경하는 난독화 기술이다. 이 난독화 기술은 가장 널리 쓰이는 난독화 기술 중의 하나로, 주로 while 구문과 switch-case 구문을 이용하여 함수 내에 존재하는 명령들의 실행 순서를 숨긴다. 그림 3.1에서 CFF 난독화의 간단한 예를 보인다[3]. CFF 난독화는 두 방법으로 구현되었는데, 함수에 등장하는 명령들 전체를 대상으로 CFF를 적용하는 방법과 함수 일부를 임의로 선정하여 CFF를 적용하는 방법이다. 편의상 앞의 방법을 CFF로, 뒤의 방법을 wCFF(weak CFF)로 표기한다.

Call site unification(CSU) 함수들간의 호출 구조를 알아보기 어렵도록 변경하는 난독화이다. 함수의 호출 관계는 프로그램 분석에 있어서 매우 중요한 정보이다. 이 난독화 기술은 함수 호출 관계를 숨기기 위해 일반적인 함수 호출 과정을 미리 정해진 여러 단계를 거치도록 변경한다. 이 실험에서는 프로그램 내의 모든 함수 호출이 동일한 지점을 통해서 이루어지도록 프로그램을 변경한다.

그림 3.1 CFF 난독화의 예

```
1  /* Original Code */
2
3  i = 1;
4  s = 0;
5  while (i < 100) {
6    s += i;
7    i++;
8  }
```

```
1  /* Obfuscated Code */
2
3  swVar = 1;
4  while (swVar != 0) {
5    switch(swVar) {
6      case 1:
7        i = 1; s = 0; swVar = 2;
8        break;
9      case 2:
10       if (i < 100) swVar  = 3;
11       else swVar = 0;
12       break;
13     case 3:
14       s += i; i++; swVar = 2;
15       break;
16   }
17 }
```

이 난독화가 적용되면 모든 함수가 자신을 포함한 모든 함수를 호출할 수 있는 것처럼 함수 호출 그래프가 변경된다. CSU 난독화의 간단한 예와 bbe-0.2.2에 CSU 난독화를 적용했을 때의 함수 호출 그래프를 각각 그림 3.2와 3.3에 나타내었다.

3.2 실험 결과

실험을 통해 계산된 점수들을 표 3.2에서 확인할 수 있다. 표 3.2의 각 샘플 프로그램별 복잡도 점수를 얻는 과정에 대한 예시로, bbe-0.2.2에 대한 실험 결과와 계산 과정을 표 3.3에 나타내었다. 실험으로 얻어진 결과가 기존에 알려진 난독화 기술들 간의 우열과 일치하는지 확인해 봄으로써 이 평가 방법이 난독화 기술 평가에 적합한지 확인해 볼 수 있다. 난독화 기술들간의 알려진 우열 관계는 다음과 같다.

- 단순히 프로그램에서 보여지는 표현 방법만을 바꾸는 난독화는 무의미하다. 앞에서 프로그램의 의미를 더 복잡하게 변경하는 난독화 기술을 더 좋은 난독화기술이라고 정의하였다. 그러므로, 단순히 문자열의 표현 방식을 변경하거나 변수의 이름을 치환하는 것은 프로그램의 의미에 어떤 영향도 주지 못하므로 이러한 난독화 기술들은 의미없는 난독화 기술로 간주할 수 있다.

- 동일한 난독화 기술이라면 더 폭넓은 영역에 적용될수록 더 강력한 난독화를 제공한다. 난독화가 적용되는 영역이 넓어질수록 프로그램의 전체적인 복잡도는 더 증가하기 때문에 더 강력한 난독화가 적용된다고 할 수 있다.

- 다양한 난독화 기술을 조합하여 적용하면 더 강력한 난독화가 가능하다. 예를 들어, 변수의 값을 숨기는 난독화와 제어 흐름을 숨기는 난독화 기술이 동시에 적용된 프로그램은 둘 중의 한 가지 난독화 기술만 적용된 프로그램보다 더 복잡한 프로그램이 된다.

그림 3.2 CSU 난독화의 예

```
1  /* Original Code */
2
3  void f() { g(); }
4  void g() { return; }
5
6  int main() {
7    f();
8  }
```

```
1  /* Obfuscated Code */
2
3  int callfunc(int cmd);
4
5  void f() { callfunc(1); }
6  void g() { return; }
7
8  int callfunc(int cmd) {
9    switch (cmd)
10   case 0: f(); return;
11   case 1: g(); return;
12 }
13
14 int main() {
15   callfunc(0);
16 }
```

그림 3.3 CSU 난독화로 인한 함수 호출 그래프의 변화(bbe-0.2.2)

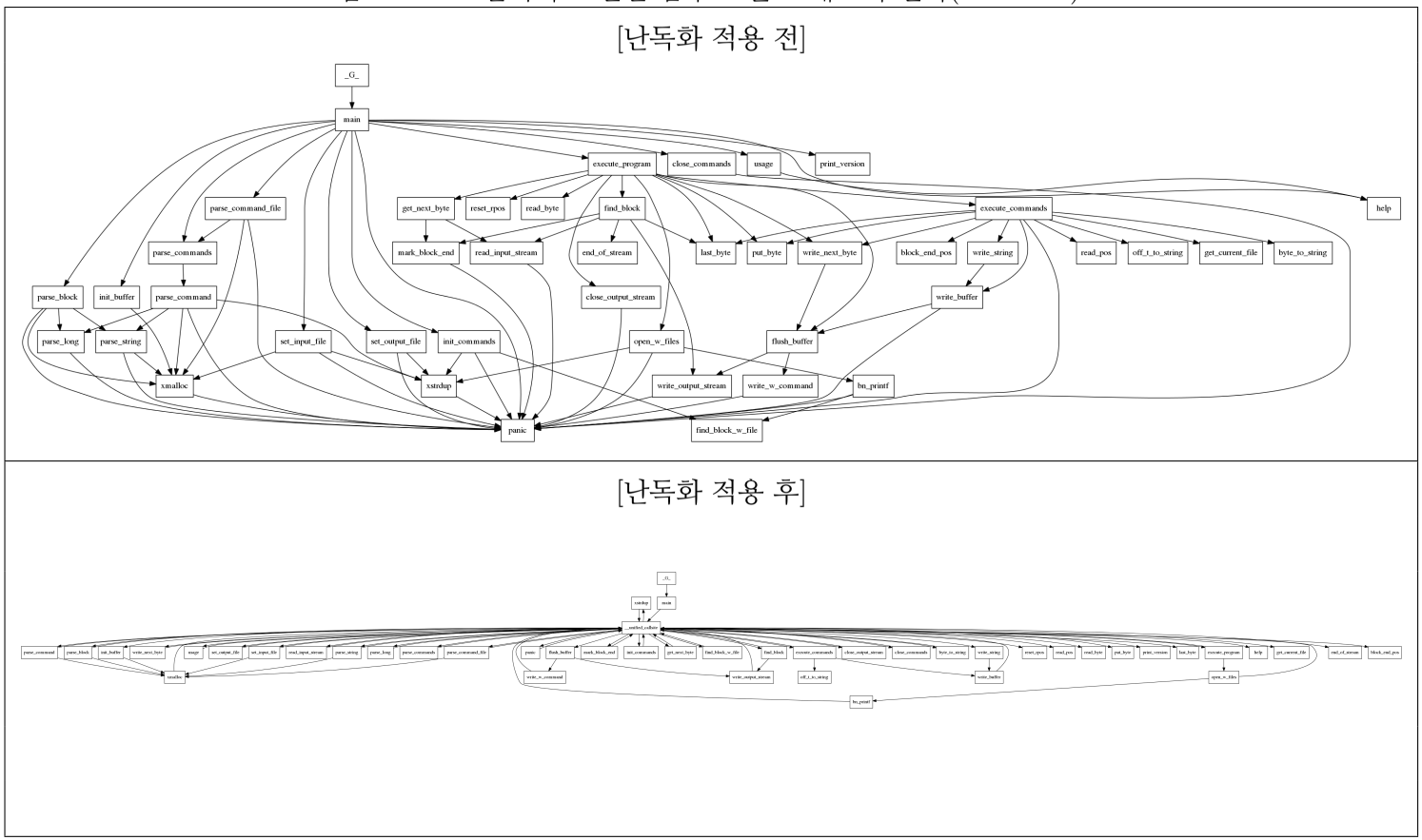
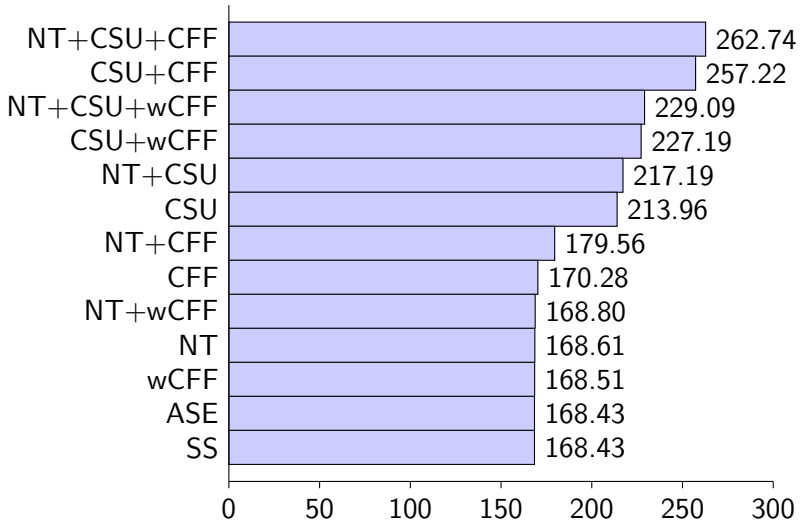


표 3.2 난독화 기술들에 대한 복잡도 점수와 복잡도 순서

Obfuscation Technique	Sample Programs			Complexity Score
	bbe	dancer-xml	xfpt	
ASE	168.80	170.12	166.36	168.43
SS	168.80	170.12	166.36	168.43
NT	168.98	170.12	166.72	168.61
CSU	208.32	213.64	219.91	213.96
CFF	172.91	170.17	167.77	170.28
wCFF	168.94	170.12	166.47	168.51
NT+CSU	208.10	217.17	226.29	217.19
NT+CFF	188.08	170.29	180.30	179.56
NT+wCFF	169.37	170.13	166.91	168.80
CSU+CFF	240.37	264.05	267.25	257.22
CSU+wCFF	226.67	224.93	229.98	227.19
NT+CSU+CFF	283.18	264.95	240.08	262.74
NT+CSU+wCFF	227.49	224.20	235.58	229.09



실험을 통한 평가 결과는 위에서 얘기한 난독화에 대한 알려진 우열 관계와 일치한다. 단순히 변수 명을 바꾸거나 문자열 표현 방식을 바꾸는 난독화 기술은 가장 낮은 점수를 얻었다. 모든 가능한 경우에서 CFF 난독화가 포함된 난독화는 CFF 대신 wCFF가 포함된 난독화보다 높은 점수를 획득하였다. 또한, 난독화 기술이 조합되어 적용되었을 때 가능한 부분 조합들보다 항상 더 높은 점수를 받은 것을 확인할 수 있다.

표 3.3 bbe-0.2.2에 대한 복잡도 계산 과정

Obfuscation	Dense						Sparse			Selective			Score
Technology	Time	Rate	T-Score	LEH	Rate	T-Score	Time	Rate	T-Score	LEH	Rate	T-Score	
original	8.76			569635		2.26			52924				
ASE	8.81	0.005	40.42	569635	0.000	42.20	2.41	0.064	42.01	52924	0.000	44.18	168.80
SS	8.73	-0.003	40.42	569635	0.000	42.20	2.36	0.045	42.01	52924	0.000	44.18	168.80
NT	11.71	0.337	40.46	697663	0.225	42.28	3.06	0.351	42.01	123359	1.331	44.23	168.98
CSU	892.33	100.825	53.29	15694077	26.551	51.49	1223.32	540.055	52.17	9281777	174.379	51.37	208.32
CFF	170.58	18.465	42.78	3219037	4.651	43.83	16.48	6.291	42.12	68754	0.299	44.19	172.91
CFFs	12.12	0.383	40.47	693369	0.217	42.28	3.34	0.477	42.01	59070	0.116	44.18	168.94
NT+CSU	792.84	89.472	51.84	16139292	27.333	51.76	1328.41	586.536	53.05	9387879	176.384	51.45	208.10
NT+CFF	859.53	97.082	52.81	6643027	10.662	45.93	92.05	39.711	42.75	3144800	58.421	46.58	188.08
NT+CFFs	18.69	1.133	40.56	972855	0.708	42.45	4.62	1.045	42.02	250565	3.734	44.33	169.37
CFF+CSU	2049.46	232.866	70.14	40964650	70.914	67.02	1268.08	559.854	52.55	8386755	157.468	50.67	240.37
CFFs+CSU	1134.31	128.437	56.81	17206401	29.206	52.42	2864.21	1265.797	65.84	9581473	180.042	51.60	226.67
NT+CFF+CSU	1567.76	177.899	63.12	51018313	88.563	73.19	2836.86	1253.699	65.61	47643245	899.220	81.26	283.18
NT+CFFs+CSU	1139.69	129.051	56.89	18075568	30.732	52.95	2866.42	1266.775	65.86	9822861	184.603	51.79	227.49

제 4 장 고찰

정적 분석에 기반하여 난독화 기술의 영향을 비교평가할 수 있는 방법을 제안하였다. 안전한 분석기를 사용함으로써 표준 점수에 기반하여 완전히 다른 종류의 난독화 기술들과 그들의 조합에 대한 복잡도를 수치화 할 수 있었고, 실험을 통해 얻어낸 복잡도 점수들이 기존에 알려진 난독화 기술의 우열과 일치함을 확인할 수 있었다.

계산된 복잡도 점수가 난독화 기술의 복잡도를 완전히 정확하게 표현하지는 못하지만 비교를 위한 기준이 될 수는 있다. 이 아이디어는 우리가 일반적으로 접하는 CPU나 비디오 카드 등에 대한 벤치마크의 아이디어와 동일하다. 이들 벤치마크는 샘플 프로그램을 실행시키고 연산 속도나 FPS(frame per second) 값 등을 측정하는 방식으로 간접적인 성능 평가를 실시한다. 이렇게 획득한 벤치마크 점수는 성능에 대한 절대적인 지표는 아니지만 성능을 비교하는 기준으로 활용된다. 이 연구에서 제안한 방법도 난독화된 프로그램에 대한 정적 분석을 통해 난독화 기술을 간접적으로 평가하기 때문에 난독화 기술의 복잡도에 대한 절대적인 기준을 제시하지는 못하지만 난독화 기술을 비교하는데에는 충분히 활용할 수 있다.

이로써 난독화 기술에 대한 종합적인 평가가 가능해졌다. 난독화 기술을 종합적으로 평가하기 위해서는 난독화 기술의 영향 뿐만 아니라 난독화 기술이 프로그램의 궁극적인 의미를 잘 유지하는지, 프로그램의 성능을 얼마나 감소시키는지에 대한 고려 등이 수반되어야 한다. 변환된 프로그램이 원본 프로그램과 동일한 동작을 하는지는 논리적으로 증명이 가능하고, 프로그램의 성능을 측정하기 위해 소프트웨어 공학적인 기법들이 이미 많이 나와 있기 때문에 본 연구에서 제안한

방법과 함께 사용하면 난독화 기술에 대한 종합적인 평가가 가능하다.

제안한 방법이 완벽한 방법은 아니다. 무엇보다, 이 시스템은 난독화 기술의 복잡도를 완전히 정확하게 표현하지 못하는 한계가 있다. 단적인 예로, 프로그램 분석에 대한 전문적인 지식이 있는 사람은 이 평가 시스템을 속여서 실제로는 매우 간단하지만 높은 평가를 받는 난독화 기술을 만들 수 있다. 또, 프로그램 분석기 때문에 발생하는 제약도 있다. 이 연구에서도, Sparrow가 C 소스 코드 분석기이기 때문에, 실험을 위해서 C 소스 코드를 위한 난독화기를 구현해야 했다. 마찬가지로, 바이너리를 대상으로 하는 난독화기를 평가하기 위해서는 안전한 바이너리 분석기가 필요하다.

제 5 장 결론

이 논문에서는 정적 분석 기술을 바탕으로 난독화 기술의 총체적인 영향을 정량적으로 비교평가할 수 있는 시스템을 제안하였다. 먼저, 안전한 정적 분석이 가능한 프로그램 분석기와 샘플 프로그램, 그리고 평가하고자 하는 난독화기로 구성된 평가 시스템을 제안하고, 정적 분석 결과를 바탕으로 복잡도를 계산하는 방법을 설계하였다. 이 시스템을 통해 다양한 난독화 기술들을 평가한 결과가 기존에 알려진 난독화 기술들간의 우열과 일치함을 확인함으로써, 이 시스템이 실용적인 난독화기의 평가에 사용될 수 있음을 보였다.

참고문헌

[1] Stephen Drape, "Obfuscation of Abstract Data-Types," Ph. D. Dissertation, The Universite of Oxford, 2004.

[2] X. Zhang, F. He, and W. Zuo, "Theory and Practice of Program Obfuscation," *Convergence and Hybrid Information Technologies, Marius Crisan (Ed.)*, InTech.

[3] T. László and Á. Kiss, "Obfuscating C++ Programs via Control Flow Flattening," *Annales Universitatis Scientiarum de Rolando Eötvös Nominatae—Sectio Computatorica,* pp.15 -15 2008.

[4] B. De Sutter, B. De Bus, K. De Bosschere, B. Preneel, B. Anckaert, and M. Madou, " Program Obfuscation: A quantitative approach," in *Proceedings of the 2007 ACM workshop on Quality of protection (QoP),* 2007.

[5] R. Giacobazzi and A. Toppan, "On Entropy Measures for Code Obfuscation," in *ACM SIGPLAN Software Security and Privacy Workshop*, ACM, 2011.

[6] H.Oh and K. Yi, "Access-based Abstract Memory Localization in Static Analysis" in *Science of computer programming*, 78.9, 1701-1727, 2013.

[7] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi, "Design and Implementation of Sparse Global Analyses for C-like Languages," in *ACM SIGPLAN Confer-*

ence on Programming Language Design and Implementation (PLDI), Vol. 47, No. 6, pp. 229-238. ACM, 2012

[8] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi, "Selective Context-Sensitivity Guided by Impact Pre-analysis," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp.475-484. ACM, 2014.

[9] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL)*, pp. 238-252, 1977.

[10] J. Cappaert and B. Preneel, "A General Model for Hiding Control Flow", *In Proceedings of the tenth annual ACM workshop on Digital rights management*, pp. 35-42, ACM, 2010.

[11] P. Cousot and R. Cousot, "Systematic Design of Program Analysis Frameworks," in *Proceedings of ACM Symposium on Principles of Programming Languages (POPL),* 1979. pp. 269-282.

[12] CIL(C Intermediate Language), http://kerneis.github.io/cil

[13] R. Giacobazzi, N.l D. Jones, and I. Mastroeni, "Obfuscation by Partial Evaluation of Distorted Interpreters," *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, pp. 63-72, ACM, 2012.

[14] Y. Sakabe, M. Soshi, and A. Miyaji, "Java Obfuscation with a Theoretical Basis for Building Secure Mobile Agents," *Communications and Multime-*

dia Security. Advanced Techniques for Network and Data Protection, 2003, p. 89-103.

[15] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji, "Software Obfuscation on a Theoretical Basis and Its Implementation," *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 2003, 86.1: 176-186.

Abstract

This paper proposes a method for evaluating effectiveness of obfuscation techniques based on static analysis. Industrial developers have to obfuscate their programs to prevent them from being reverse engineered, so they need an obfuscator evaluation method to select the best obfuscator. I propose a simple method to evaluate various obfuscators through comparing semantic-based static analysis results of obfuscated programs. In the experiment, I show that evaluated results offered by my method align with known effectiveness of obfuscation techniques.

Keywords: Obfuscation, Complexity Evaluation, Static Analysis

Student Number: 2013-23116